EE 524 P
Applied High-Performance GPU Computing

LECTURE 7 : Thursday, November 8, 2018

Instructor: Dr. Colin Reinhardt

University of Washington - Professional Masters Program

Autumn 2018

Lecture 7 : Outline

- HW4 Due 11/15 at 6:00 PM (NEXT THUR)
- Final Project Reminders
  - Project Proposals: due 11/15 at 6:00 PM (NEXT THUR)
  - Proposal Designs: due 11/29 at 6:00 PM
  - Final Project: due 12/14 by 6:00 PM (no late submissions)
- Parallel Performance Theory and Metrics
- Image Processing with OpenCL
  - Spatial Domain Image Processing: Image Enhancement
    - Point-based gray-level transformations
    - Sobel-Feldman gradient edge-detection filter/stencil
    - Laplacian edge-detection filter/stencil
- EX7

Final Project: Proposals

Project Proposal: 1-2 pages

MUST include:

- Summary of topic/problem to be studied
- List of primary references to be used
- Role of OpenCL in problem solution

If you still don't have a final project idea or are unsure

- email me
- come to Sunday office hour
- look at the course website page: https://canvas.uw.edu/courses/1260593/pages/final-project-ideas

Parallel Performance Theory and Metrics

Speedup: compares *latency* for solving the identical computational problem on one hardware unit ("worker") versus on *P* workers (cores/processors)

$$S_P = \frac{T_1}{T_P}$$

T_1 is the latency of program with one worker

T_P is the latency of program with *P* workers

Efficiency: ratio of speedup to the number of workers

$$\text{Efficiency} = \frac{S_P}{P} = \frac{T_1}{PT_P}$$

Relative speedup: uses serialization of the parallel algorithm as baseline for T_1

Absolute speedup: uses a better serial algorithm for baseline that may not parallelize well

- Both algorithms must solve the identical computational problem

Linear speedup: algorithm runs P times faster with P processors

Superlinear speedup: efficiency > 100%

Scalability

- **Strong scaling**
  - Considers speedup as *P* varies and the problem size remains fixed.
    - Amdahl's Law
- **Weak scaling**
  - Assumes the problem size grows proportionally with *P*.
    - Gustafson-Barsis' Law

Example

Assume problem size is *M* (the working set in main memory).

There are *P* processors. What is the memory per processor, for

Strong scaling?

Weak scaling?

Asymptotic Speedup and Efficiency

Example: dot-product of two vectors of length N with P workers (assume $P \leq N$)

- Partition vectors between the P workers
- Each worker computes sub-dot product of length N/P
- Sum sub-products using tree pattern with height $= \log_2 P$

- Asymptotic speedup: $S_P = \frac{T_1}{T_P} \approx \frac{\Theta(N)}{\Theta(N/P + \log_2 P)} = \Theta\left(\frac{N}{N/P + \log_2 P}\right)$

  If $N \gg \log_2 P$ then $S_P \approx \Theta(P)$

- Asymptotic efficiency AE $= \frac{T_1}{P \cdot T_P} \approx \Theta\left(\frac{N}{N + P \log_2 P}\right)$

  If $N \approx \Theta(P \log_2 P)$ then AE $\approx \Theta(1)$

Parallel Performance Theory and Metrics

Execution time T_1 of a program can be divided into

W_{ser} : time spent doing non-parallelizable serial work

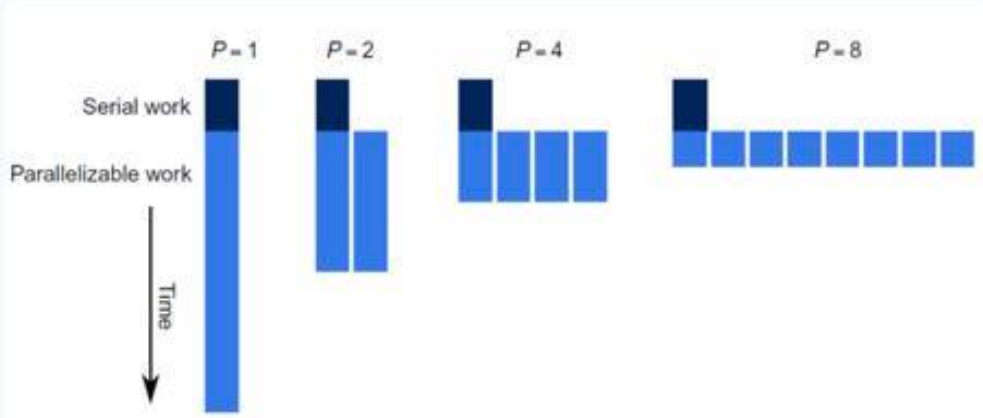
W_{par} : time spent doing parallelizable work

Then given *P* workers

$$T_1 = W_{ser} + W_{par} \qquad \text{and} \qquad T_P \geq W_{ser} + \frac{W_{par}}{P}$$

Amdahl's Law: [Gene Amdahl, 1967]

$$S_P \leq \frac{T_1}{T_P} = \frac{W_{ser}+W_{par}}{W_{ser}+\frac{W_{par}}{P}}$$



Parallel Performance Theory and Metrics

Amdahl's Law: Corollary

Let f be the non-parallelizable serial fraction of the total work.

Then the following inequalities hold:

$$W_{ser} = fT_1$$
$$W_{par} = (1-f)T_1$$

Substituting into Amdahl's Law gives $$S_P \leq \frac{1}{f+\frac{1-f}{P}}$$

When *P* tends to infinity $$S_\infty \leq \frac{1}{f}$$

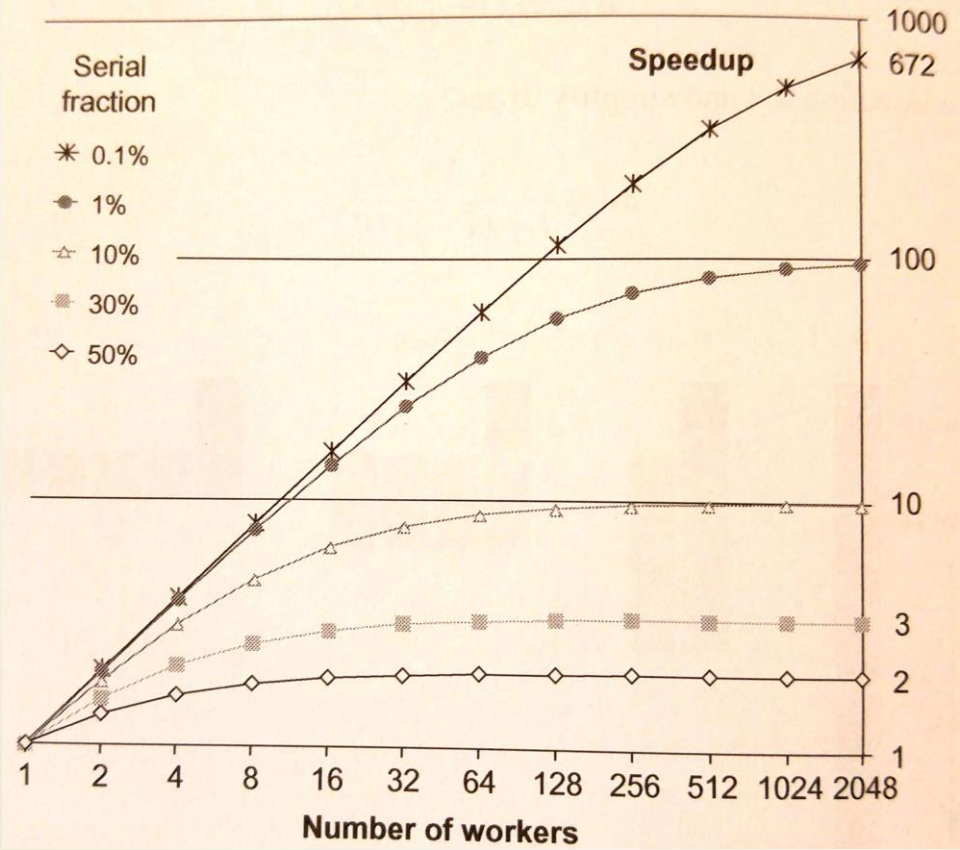
Speedup is fundamentally limited by the fraction of work that is not parallelizable, even with an infinite number of processors.

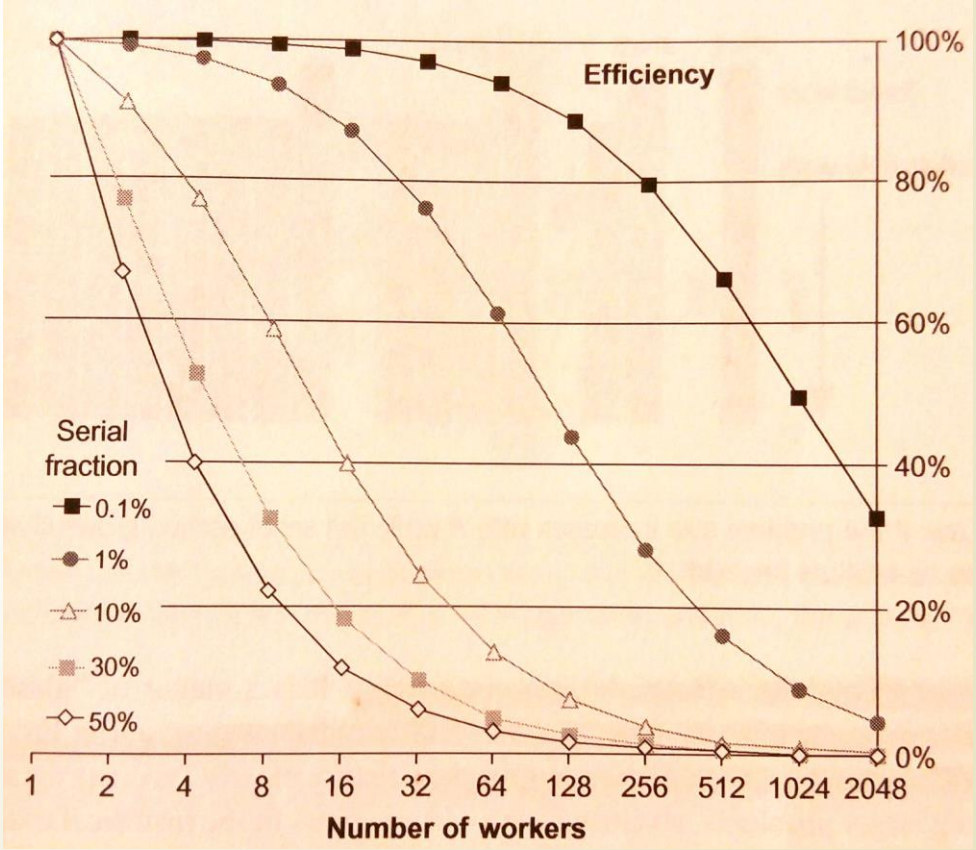
Parallel Performance Theory and Metrics

Amdahl's Law

Views program size as fixed and computers as changeable

speedup fundamentally limited by serial fraction



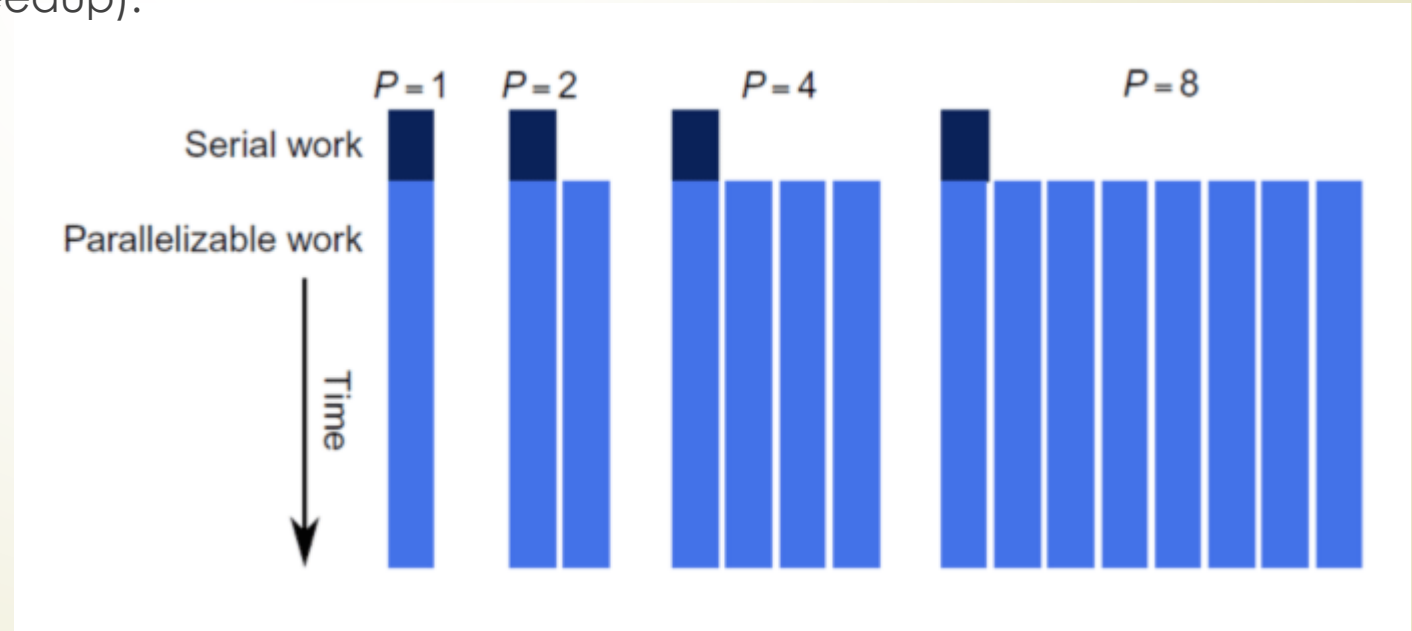


Parallel Performance Theory and Metrics

Gustafson-Barsis' Law [John Gustafson, 1988]

Noted that problem sizes grow as computers become more powerful.

- Assume serial portion is constant while parallel portion grows linearly with the problem size.
- Serial portion takes same amount of time to execute, but diminishes as a fraction of the whole.
- Once serial portion becomes insignificant, speedup grows at same rate as number of workers (~linear speedup).



Parallel Performance Theory and Metrics

Karp-Flatt Metric [Alan Karp, Horace Flatt, 1990]

- Defines the *experimentally-determined serial fraction* $\equiv f_e = \frac{W_{ser}}{T_1}$

$$f_e \leq \frac{\frac{1}{S_{P\_meas}} - \frac{1}{P}}{1 - \frac{1}{P}}$$

- actual measurement accounts for load imbalance and overhead

Follows from Amdahl's Law $T_P \geq W_{ser} + \frac{W_{par}}{P}$

$$T_P \geq f_e T_1 + \frac{T_1(1-f_e)}{P} \rightarrow \frac{T_P}{T_1} \geq f_e + \frac{(1-f_e)}{P} \rightarrow \frac{1}{S_P} \geq + \frac{Pf_e + (1-f_e)}{P}$$

$$\rightarrow \frac{P}{S_P} \geq Pf_e + (1-f_e) \rightarrow \frac{P}{S_P} \geq 1 + f_e(P-1) \rightarrow \frac{\frac{P}{S_P} - 1}{(P-1)} \geq f_e$$

Spatial-Domain Image Processing

Image Enhancement with OpenCL

Spatial Domain Processing Concepts

- Spatial domain methods operate directly on the aggregate of pixels composing an image.
- In general spatial domain processes will be denoted

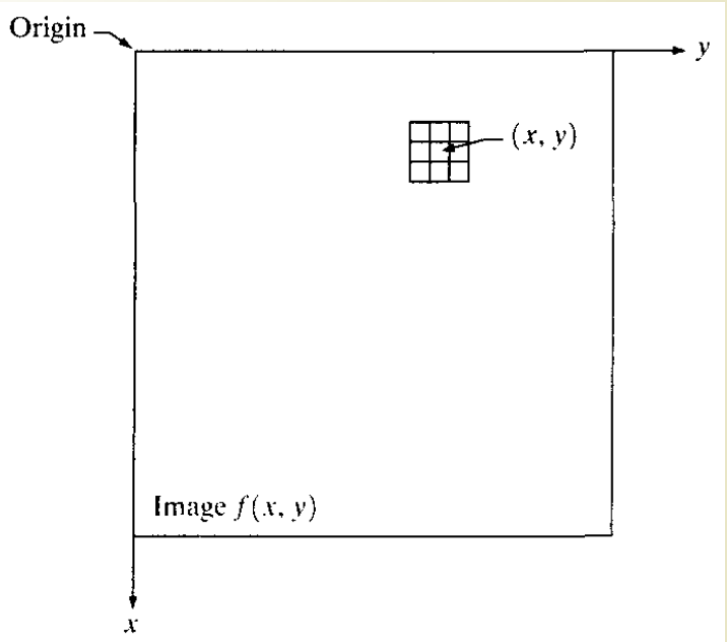
$$g(x,y) = T[f_n(x,y)]$$

output (processed) image

operator on *f*, defined over some neighborhood of *(x,y)*

input image # n

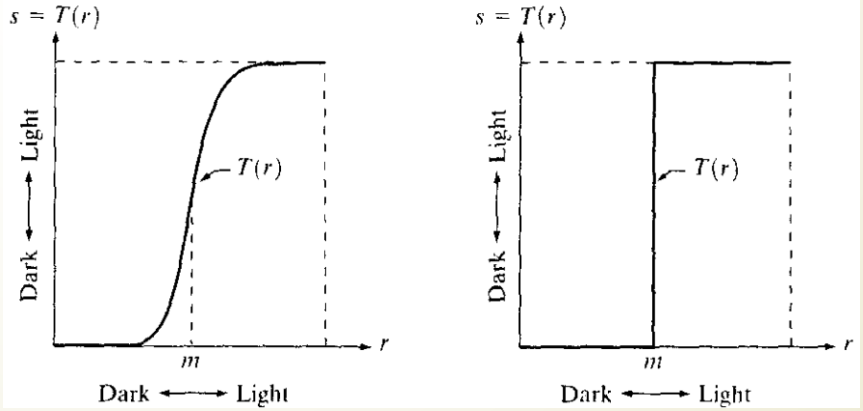
- square/rectangular subimage area (stencil)
- centered at (x,y)
- moved from pixel to pixel
- operated T applied at each location (x,y)
- yields output g(x,y)
- only local neighborhood spanned contributes



Point-Processing / Gray-Level Transformations

- Simple: $s = T(r)$
  - point-processing: neighborhood of T is 1x1 (gray-level transformations)
    - Image negatives
      - image with pixel gray levels in range r = [0, L-1]
      - negative transformation: s = L-1 - r
    - Log Transformations
      - s = c *log*(1 + r), for r >= 1
    - gamma-correction and contrast enhancement (power-law transformations)
      - $s = cr^{\gamma}$, where c and γ are positive constants

contrast stretching



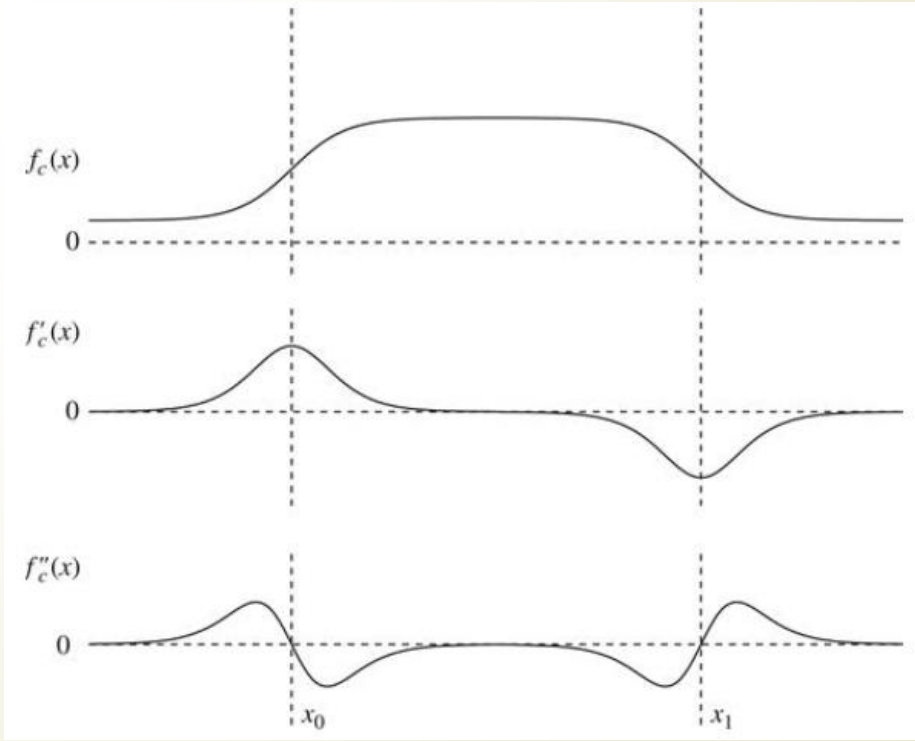
thresholding

- Larger neighborhoods allow considerably more flexibility
  - Values of the 2D mask/filter/stencil coefficients determine nature of process

Image Processing Case Study

Edge Detection Filter (stencil)

- Edge Detection: a fundamental image analysis operation
  - Edges provide vital information in analysis and interpretation of image information, both for biological vision and computer vision/image analysis.
  - Mathematical gradient and Laplacian (zero-crossings) can be used to identify edge locations



Gradient-based Edge Detection Stencil

- Recall the gradient vector: $\nabla f_c(x,y) = \frac{\partial f_c(x,y)}{\partial x}\boldsymbol{i}_x + \frac{\partial f_c(x,y)}{\partial y}\boldsymbol{i}_y$
- A discrete differential operator, computes (relatively crude) approximation of gradient of the image intensity function

$$\hat{\nabla} f_c(n_1,n_2) = f(n_1,n_2) * h_1(n_1,n_2)\boldsymbol{i}_x + f(n_1,n_2) * h_2(n_1,n_2)\boldsymbol{i}_y$$

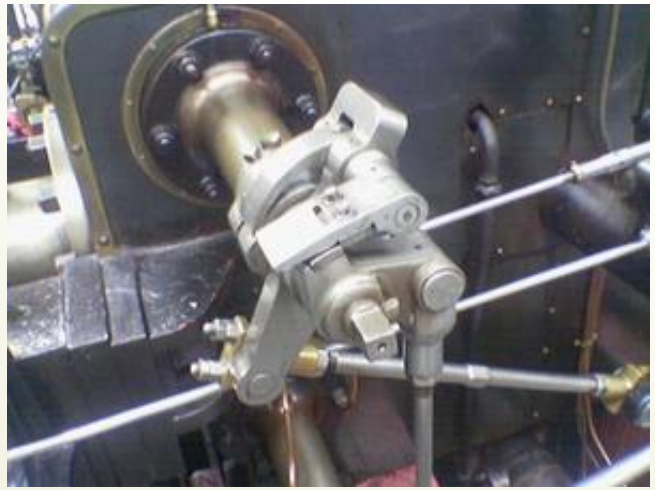
- 1D central-difference scheme: $\frac{df}{dx} \cong \frac{f(x+\Delta x)-f(x-\Delta x)}{2\Delta x} \rightarrow \frac{f(n+1)-f(n-1)}{2}\Big|_{\Delta n=1}$
- which can be written $\frac{df}{dx} = f(x) * h(x)$
  - where $h(x) = \frac{1}{2}\big(\delta(x-1) - \delta(x+1)\big) \rightarrow h[n] = \frac{1}{2}[1 \quad 0 \quad -1]$
  - 2D central differences: (horizontal) $\begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & -1 \\ 0 & 0 & 0 \end{bmatrix}$ and (vertical) $\begin{bmatrix} 0 & -1 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$
- noise-sensitivity of first-order difference operators reduced by smoothing in orthogonal direction: $h_a(n_1)h_b(n_2) = h_1(n_1,n_2)$
  - outer-product forms separable derivative-smoothing filter $\begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix}[1 \quad 0 \quad -1] = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$

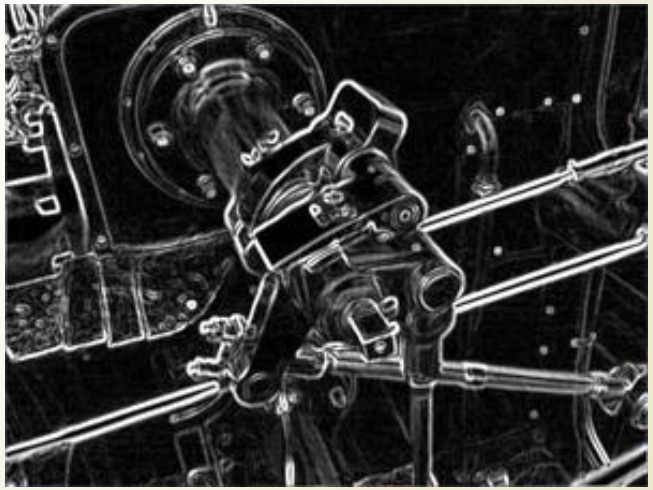
Sobel-Feldman Gradient Stencil

- Discrete approximation takes form of a pair of orthogonally-oriented filters
- Uses two 3x3 separable integer-valued filter kernels convolved with image

$$\mathbf{G}_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} * \mathbf{A} \quad \text{and} \quad \mathbf{G}_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * \mathbf{A}$$

- **Gradient magnitude** is computed using $|\nabla \boldsymbol{G}| = \sqrt{\boldsymbol{G}_x^2 + \boldsymbol{G}_y^2}$





Due to computational burden often use: $|\nabla \boldsymbol{G}| \approx |G_x| + |G_y|$

Second-order Edge Detection

- **Laplacian** $\nabla^2 f = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}$
- the simplest isotropic linear derivative operator
  - response is independent of the direction of the discontinuities in the image
  - rotation invariant
- For digital image processing, need discrete 2D representation

$$\frac{\partial^2 f}{\partial x^2} \approx f(x+1,y) - 2f(x,y) + f(x-1,y)$$

$$\frac{\partial^2 f}{\partial y^2} \approx f(x,y+1) - 2f(x,y) + f(x,y-1)$$

- Summing these components gives discrete 2D Laplacian

$$\nabla^2 f \approx f(x+1,y) + f(x-1,y) + f(x,y+1) + f(x,y-1) - 4f(x,y)$$

Laplacian Edge Detection Stencils

$$\nabla^2 f(x,y) \approx f(x+1,y) + f(x-1,y) + f(x,y+1) + f(x,y-1) - 4f(x,y)$$

- Using this expression gives the digital Laplacian filter stencil

f(x-1,y-1)	f(x,y-1)	f(x+1,y-1)
f(x-1,y)	f(x,y)	f(x+1,y)
f(x-1,y+1)	f(x,y+1)	f(x+1,y+1)

0	1	0
1	-4	1
0	1	0

- Add diagonal terms:

1	1	1
1	-8	1
1	1	1

Laplacian Edge Detection Stencils

$$\nabla^2 f(x,y) \approx f(x+1,y) + f(x-1,y) + f(x,y+1) + f(x,y-1) - 4f(x,y)$$

- Laplacian operator highlights gray-level discontinuities and deemphasizes regions of slow-varying gray level.
- recover background features while preserving sharpening effects

$$g(x,y) = f(x,y) - \nabla^2 f(x,y)$$

- By substituting the discretized Laplacian expression we get

0	-1	0
-1	5	-1
0	-1	0

and

-1	-1	-1
-1	9	-1
-1	-1	-1

- (known as ***composite Laplacian*** masks/filters/stencils)

In-class Exercise 7a

- 1st order (gradient) and 2nd order (Laplacian) Edge Detection Kernels
- See procedures on class website /InclassExercises/Ex7

Combining Spatial Enhancement Filters

- Often image enhancement tasks require application of several complementary techniques to achieve a desired result.

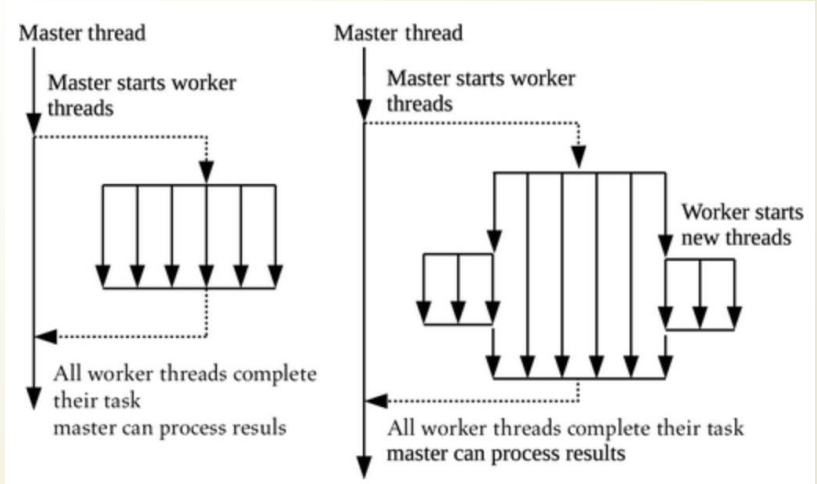
- Suppose we wanted to apply
  - a Sobel gradient edge-detection filter
  - then a blur-kernel filter
  - followed by a Laplacian edge-sharpening filter
  - followed by an image rotation

- We don't want to transfer the image back-and-forth multiple times between Host (CPU) and device (GPU)
  - this is MUCH worse on a discrete GPU card (i.e. AMD or NVIDIA)
- Ideally we could leave the intermediary image on the device and apply multiple consecutive kernels before retrieving result on Host.

Device-Side Queuing

- We can do this using new OpenCL 2.0 feature: **Device-Side Queuing**

- Allow a *child kernel* to be enqueued directly from a kernel executing on a device (the *parent kernel*) independently from the Host
- Enables *nested parallelism* model
  - parallel programming paradigm where a thread can spawn additional threads to execute additional tasks.
  - commonly occurs in applications where number of threads required to execute tasks is not known in advance



Device Enqueue

- In order for a parent kernel to enqueue a child kernel it will call the OpenCL C built-in function `enqueue_kernel( ).`
  - `enqueue_kernel( )` will enqueue a kernel for each work-item that executes function!

```
int enqueue_kernel( queue_t queue,
                    kernel_enqueue_flags_t flags,
                    const ndrange_t ndrange,
                    void (^block)(void) )

int enqueue_kernel( queue_t queue,
                    kernel_enqueue_flags_t flags,
                    const ndrange_t ndrange,
                    uint num_events_in_wait_list,
                    const clk_event_t *event_wait_list,
                    clk_event_t *event_ret,
                    void (^block)(void) )
```

  - there are 2 other variants of `enqueue_kernel()` which allow allocating "dynamic" local memory within the child kernel (see OpenCL C Spec, section 6.13.17)

Device Enqueue

Host application modifications:

- Host will need to create a separate command-queue for the device
  - it can either be set as a default queue for the device,
  - or passed as an argument to the parent kernel
- Here is example of creating a default queue for the device (on the host)

```
const cl_queue_properties queueProps[] = { CL_QUEUE_PROPERTIES, (cl_command_queue_properties)CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE
|                                                                CL_QUEUE_ON_DEVICE | CL_QUEUE_ON_DEVICE_DEFAULT, 0 };
cl_command_queue commandsDev = clCreateCommandQueueWithProperties(context, device_id, queueProps, &err);
```

- Inside the parent kernel, it can then obtain the default queue using

```
queue_t q = get_default_queue();
```

enqueue_kernel(…)

- *flags* parameter
  - specifies when child kernel should begin executing
  - **CLK_ENQUEUE_FLAGS_NO_WAIT**: The child kernel can begin executing immediately.
  - **CLK_ENQUEUE_FLAGS_WAIT_KERNEL**: The child kernel must wait for the parent kernel to reach the ENDED before executing. In this case, the parent kernel has finished executing. However, other child kernels could still be executing on the device.
  - **CLK_ENQUEUE_FLAGS_WAIT_WORK_GROUP**: The child kernel must wait for the enqueuing work-group to complete its execution before starting
- *ndrange* parameter
  - built-in types and functions available

```
size_t globalSize[2] = {imgWidth,imgHeight};
size_t localSize[2] = {32, 2};
ndrange_t myNDrange2D = ndrange_2D(globalSize,localSize);
```

Device Enqueue

- Great!! … but what is `void (^block)(void)`??
  - Clang Block syntax
  - child kernels are enqueued as code represented by Block syntax
- Very simple example

```
kernel void child_kernel(uint parent_gid)
{
   uint child_gid = get_global_id(0);
   printf("<child_kernel> Parent gid: %d, Child gid: %d\n",parent_gid,child_gid);
}

kernel void parent_kernel(void)
{
   uint pgid = get_global_id(0);
   queue_t q = get_default_queue();

   printf("<parent_kernel> Parent gid: %d.  BEFORE enqueuing child kernels...\n",pgid);
   enqueue_kernel(q, CLK_ENQUEUE_FLAGS_NO_WAIT, ndrange_1D(8,2), ^{ child_kernel(pgid); });
   printf("<parent_kernel> Parent gid: %d. AFTER enqueuing child kernels...\n",pgid);
}
```

- refer to OpenCL C Spec, section 6.13.17.1 for more examples

Device Enqueue

- Device-side kernel synchronization!
- Just as host-side commands have events, so do device-side commands
  - Every command enqueued can return an event object
    - `clk_event *event_ret`
  - Every command enqueued can specify an event wait list of events it depends on, which must complete before it will begin executing
    - `uint num_events_in_wait_list`
    - `const clk_event *event_wait_list`
- We'll need to use these to synchronize execution between our child kernels to ensure correct behavior.

Combining Spatial Enhancement Methods

- OpenCL Design and implementation
  - we will use the device-side enqueue mechanism to achieve our goals.
  - we'll use another new pattern which is the parent scheduler kernel : "master"
    - launch parent kernel as NDRange dimension = global_size = local_size = 1
    - swap image objects and event handles between consecutive child kernel enqueues

Multiple filter device-side enqueue kernel pseudocode

```
__kernel void filter1(__read_only image2d_t inImg, __write_only image2d_t outImg, … ) { … };
__kernel void filter2(__read_only image2d_t inImg, __write_only image2d_t outImg, … ) { … };
__kernel void filter3(__read_only image2d_t inImg, __write_only image2d_t outImg, … ) { … };

__kernel void master(__read_write image2d_t inImg, __read_write image2d_t outImg, … /* all parent & child args */)
{
     // get default queue, setup NDRange, define event handles…

     enqueue_kernel(q, CLK_ENQUEUE_FLAGS_NO_WAIT, myNDrange, 0, NULL, &evt1, ^{ filter1(inImg,outImg, …); } );
     enqueue_kernel(q, CLK_ENQUEUE_FLAGS_NO_WAIT, myNDrange, 1, &evt1, &evt2, ^{ filter2(outImg,inImg, …); } );
     enqueue_kernel(q, CLK_ENQUEUE_FLAGS_NO_WAIT, myNDrange, 1, &evt2, NULL, ^{ filter3(inImg,outImg, …); } );
     // repeat as needed …
}
```

In-class Exercise 7b

- Combining spatial image enhancement methods
- OpenCL 2.0 device-side enqueuing
- See procedures on class website /InclassExercises/Ex7